Translating Forth to Efficient C

M. Anton Ertl Martin Maierhofer

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
anton@mips.complang.tuwien.ac.at, m.maierhofer@ieee.org
Tel.: (+43-1) 58801 4474
Fax.: (+43-1) 505 78 38

Abstract. An automatic translator can translate Forth into C code which the current generation of optimizing C compilers compiles to efficient machine code. I.e., the resulting code keeps stack items in registers and rarely updates the stack pointer. This paper presents a simple translation method that produces efficient C code, describes an implementation of the method and presents results achieved with this implementation: The translated code is 4.5–7.5 times faster than Gforth (the fastest measured interpretive system), 1.3–3 times faster than BigForth 386 (a native code compiler), and smaller than Gforth's threaded code.

1 Introduction

Due to C's popularity, there are many sophisticated optimizing compilers for C. Many language implementations (e.g., Eiffel) have taken advantage of this fact and use C as intermediate language in their compilation process, i.e., they use C as "portable assembler".

In the same way, C can be used as intermediate language for Forth native code compilation. While this approach requires sacrificing some of Forth's features (mainly interactivity), and is therefore unlikely to become the dominant Forth implementation method, it is useful in the following contexts:

- The customer may require C.
- It can serve as a replacement for the practice of recoding time-critical words in assembly language. The advantages of this approach are that the translation is automatic and the result is machine-independent. However, it requires the ability to call C functions from Forth.
- It can be used for evaluating native code compilers, by comparing their code with the code produced by going through C.
- It can be used for evaluating the usefulness of certain optimization methods with respect to existing Forth source code, by comparing the speed of the code resulting from compiles with and without the optimization.

In the context of the RAFTS project for native code compilation [Ert92], we are interested primarily in the last two applications.

2 Previous Work

Several interpretive Forth implementations have been written in C (Cforth, TILE, PFE, ThisForth) and its relatives (Gforth in GNU C [Ert93], UNTIL in C++ [Smi92]). There are also many native code implementations, and several papers about them [Ros86, Alm86, Pay91]. However, the combination of these ideas, i.e., using C as intermediate language in native code generation has not seen many implementations; the only one we know of (apart from ours) comes with Rob Chapman's Timbre/botForth system[1].

Timbres Forth-to-C translator creates code similar in spirit to the output of a subroutine threaded Forth system that inlines selected primitives, except that it creates C source code, not machine code. It also employs a kind of peephole optimization, but not on the output, but on the Forth input: Like ThisForth, it has a mechanism for recognizing certain Forth sequences, and it has special C code generation rules for these sequences.

```
: max ( n1 n2 -- n )
 2dup < if
   swap drop
 else
   drop
 endif ;
```

Fig. 1. `max` in Forth

Figure 1 shows a definition of `max` in Forth. Figure 2 shows the C code that the Forth-to-C translator of Timbre V.4 produces for this definition. Unfortunately, current C compilers produce pretty

[1] Available at http://taygeta.oc.nps.navy.mil/pub/Forth/Reviewed/timbre.zip

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth '95 Conference Proceedings

```
void MAX()  /* N1 N2 -- N */
{
  --sp,sp[0]=sp[2];  /* OVER */
  --sp,sp[0]=sp[2];  /* OVER */
  if((Integer)sp[1]<(Integer)sp[0])
    *++sp=-1; else *++sp=0;  /* < */
  if(*sp++)  /* IF */
  {
    sp[1]=sp[0],sp++;  /* SWAP DROP */
  }
  else  /* ELSE */
  {
    sp++;  /* DROP */
  }
}
```

Fig. 2. max translated to C by Timbre

slow assembly code from this C code; e.g., Fig. 3 contains the code produced with `gcc-2.6.3 -O3 -fomit-frame-pointer` for the Intel architecture[2]. Several features of the C code cause the slow machine code; they all have to do with the inability of C compilers to disambiguate memory references (even if they try, they have little success).

The biggest problem in the code above is that the stack pointer is a global variable. In the absence of analysis of the whole program, the compiler has to assume that the address of this variable has been taken and that every store to memory through a pointer can be a store into the stack pointer, and every read from memory through a pointer could be a read from the stack pointer. So it has to keep the stack pointer in memory up-to-date nearly at all times, and it has to load it from memory after every store through a pointer. A simple way of helping the C compiler would be to use a local stack pointer variable that is initialized from the global stack pointer at the start of the function and after calls and from which the global stack pointer is updated upon leaving the function and function calls. Then the C compiler could keep the stack pointer in a register at least some of the time.

The next problem is that stack items are accessed through pointers. Again, the C compiler often cannot determine that the stack item stored to memory has not been changed there in the meantime. so, instead of keeping the item in a register, the compiler loads it from memory again. Even if it can determine that the load is unnecessary, it can hardly ever determine that the store of the stack item is unnecessary. So, accessing stack items through pointers results in a lot of memory traffic with current C compilers.

[2] Note that, in the AT&T syntax used here, the destination operand is on the right.

```
  addl $-4,_sp
  movl _sp,%edx
  movl 8(%edx),%eax
  movl %eax,(%edx)
  addl $-4,_sp
  movl _sp,%edx
  movl 8(%edx),%eax
  movl %eax,(%edx)
  movl _sp,%edx
  movl 4(%edx),%eax
  cmpl %eax,(%edx)
  jle L7
  addl $4,_sp
  movl _sp,%eax
  movl $-1,(%eax)
  jmp L8
L7:
  addl $4,_sp
  movl _sp,%eax
  movl $0,(%eax)
L8:
  movl _sp,%eax
  addl $4,_sp
  cmpl $0,(%eax)
  je L11
  movl _sp,%edx
  movl (%edx),%eax
  movl %eax,4(%edx)
L11:
  addl $4,_sp
  ret
```

Fig. 3. Timbre's C code for max compiled to Intel assembly

Exacerbating these problems are the stack pointer changes in the C code. First of all, with the global stack pointer they result in expensive read-modify-write sequences; secondly, some C compilers can determine that two accesses through the same pointer access (or do not access) the same item, but they give up if the pointer is changed between the accesses.

3 Translation

3.1 Efficient C

First of all, there is no such thing as efficient C code per se. How efficient a piece of code is depends on the compiler that is used. E.g., the code our translation method (described later) produces gives disastrous results with compilers like PCC[3], lcc [FH91], or the GNU C compiler with optimization turned off. On the other hand, maybe one day (but not in the foreseeable future) there will be compilers

[3] PCC was the C compiler distributed with most Unix systems 15 years ago.

that compile the C code produced by Timbre into perfect machine code.

So, what we mean with efficient C code in this paper is C code that is compiled to efficient machine code by a class of compilers commonly known as globally optimizing compilers, e.g., `gcc -O`.

We make extensive use of two features of these compilers: global register allocation [Cha82], i.e., the compiler tries to put as many local variables into registers as possible[4]; and copy propagation [ASU86], i.e., optimizing away simple assignments between variables (copies) by assigning the variables to the same register. These optimizations allow us to introduce unnecessary copies and additional variables for free (but the number of simultaneously live variables is limited).

Another optimization, dead code elimination [ASU86], removes all code whose results are not used; this optimization is useful for Forth code containing `drop`, in particular sequences like `r> drop`.

3.2 Principles

On register architectures, one should keep as many values in registers as possible, and should avoid accessing stuff in memory. For Forth native code generation, this means that one should try to keep the stack items in registers. The other part of the stack manipulation overhead is the stack pointer updates; they should be avoided, too.

For direct native code generation, we proposed using sophisticated compilation techniques [Ert92]. In native code generation via C, the C compiler does most of the work. We simply have to use local variables for the stack items, and the C compiler will try to allocate them into registers. So, the translator has to keep track of which tsack items reside in which C variables.

3.3 Primitives

For the translation of primitives, we can make use of the definitions of primitives in Gforth [Ert93], e.g.,

```
+    n1 n2 -- n     core    plus
n = n1+n2;
```

The first line contains useful information, most notably, the name of the primitive and its stack effect, while the other lines (in the present case the second line) contain the C code for the primitive.

Assuming that the top stack items reside in the C variables `x5`, `x6` and `x7` (`x7` on top), the translator produces the following code for executing `+`:

[4] In optimization terminology "local" means "within a basic block", global means "within a procedure", and everything beyond is called "interprocedural".

```
{
  Cell n1=x6;
  Cell n2=x7;
  Cell n;
  n = n1+n2;
  x8 = n;
}
/* stack now: ... x5 x8 */
```

First, the names in the stack effect are declared as C variables; the input variables are initialized with the top stack items. Next comes a verbatim copy of the C code for the primitive. Finally, the result is assigned to a new variable, `x8`. We could reuse an old variable, but this can restrict register allocation and copy propagation on some compilers. This sequence looks quite bulky, but a good optimizing C compiler compiles it to one instruction, and sometimes it can even compile the C sequence for several primitives (e.g., `20 + @` on most processors) into one instruction.

A sequence of primitives can be translated simply into a sequence of translations. E.g., if we want to compile another `+`, we simply append the following code to the code above:

```
{
  Cell n1=x5;
  Cell n2=x8;
  Cell n;
  n = n1+n2;
  x9 = n;
}
/* stack now: ... x9 */
```

3.4 Definitions

A Forth colon definition is translated into a C function. In this section we consider only colon definitions with fixed stack effects. First, the stack effect of the definition is computed. The C function takes as many parameters as the definition accesses stack items. If there are several result values, they are bundled into a C struct (small structs are returned in registers by some C compilers). E.g., the C function for the word `foo ( x1 x2 x3 -- n4 n5 )` is:

```
Two_cells foo(Cell x3, Cell x2, Cell x1)
{
  Two_cells result;
  Cell x4, x5, ....;

  /* stack now: ... x1 x2 x3 */
  ... /* C code for the definition */
  /* stack now: ... x15 x16 */
  result.cell1=x16;
  result.cell2=x15;
  return result;
}
```

A call of foo would look like this:

```
/* stack now: ... x11 x12 x13 */
{
  Two_cells d=foo(x13,x12,x11);
  x15=d.cell1;
  x14=d.cell2;
}
/* stack now: ... x14 x15 */
```

The parameter order is more or less arbitrary. Having the top-of-stack first ensures that in cases like

```
: foo
  bar
  ... ;
```

the parameters need not be moved to new registers before calling `bar`. However, one can also construct cases, where having the top-of-stack last is better. It has to be determined empirically which is better for real-world Forth code.

Turning definitions into C functions is simple, but makes tricks like `r> drop exit` almost impossible to translate (but they are not ANS Forth anyway). It also has a performance impact, because most C compilers are not as optimized for reducing call overhead as much as you would like for a Forth compiler. A C compiler with automatic inlining and interprocedural register allocation should perform well, however. Alternatively, we could build a certain amount of inlining into the Forth-to-C translator.

3.5 Control Structures

ANS Forth allows the creation of arbitrary control structures using the words `IF`, `AHEAD`, `THEN`, `BEGIN`, `UNTIL`, `AGAIN`, and `CS-ROLL` [Bad90]. Since translating arbitrary control structures into structured C is hard and, in the context of sophisticated C compilers, unrewarding, we translate the control structure words into `goto`s and labels:

Forth	C
`THEN`, `BEGIN`	*label*:
`AHEAD`, `AGAIN`	`goto` *label*;
`IF`, `UNTIL`	`if (`x`==0) goto` *label*;

At control flow joins (`THEN`, `BEGIN`), we have to make sure that the respective stack items of both control flows reside in the same variables. This can be done by introducing copy operations, e.g., for a `BEGIN..UNTIL`-loop:

```
/* stack now: ... x1 x2 x3 */
label1: /* BEGIN */
...
/* stack now: ... x8 x9 x10 x11 */
x1=x8;
x2=x9;
x3=x10;
if (x11==0) goto label1; /* UNTIL */
```

This assumes, of course, that both branches to be merged have the same stack depth.

3.6 Return Stack

The return stack is handled like the data stack: Its items are kept in C variables, and the translator keeps track of which items are in which variable. E.g., the translation of `>r` looks like this:

```
/* stack: ... x5, return-stack: ... */
{
  Cell n=x5;
  x6=n;
}
/* stack: ..., return-stack: ... x6 */
```

The same method works for the floating-point stack.

3.7 Names

Forth has a different, more complex and powerful name space structure than C. Forth also allows names that are not legal C identifiers. Therefore, we cannot use the Forth names directly in the C code. A simple way out is to derive the C name from the name field address of the Forth word (e.g., by printing it in base 36).

However, in practice we prefer a name that is as close to the original as possible. This can be achieved by converting special characters in names into letter sequences and by appending some digits if that is necessary to avoid conflicts.

3.8 Locals

Locals can be translated simply into C locals. If sophisticated scoping behaviour as in, e.g., [Ert94] is desired, it is probably easiest to define the locals on the C level for the whole function and to to avoid name clashes by renaming.

3.9 Other Word Types

Variables and `CREATE`d words are translated into global (or `static`) C variables of appropriate size. E.g., `5 variable flip` translates into:

```
Cell flip[]={5};
```

When used in a definition, they are translated like primitives with similar stack effect, e.g., an occurence of `flip` is translated into

```
{
  Cell n;
  n= (Cell)flip;
  x9=n;
}
/* stack now: ... x9 */
```

DOES>-parts are translated into C functions like colon defintions—their top-of-stack parameter is the address of the word. Accordingly, using a word defined with a CREATE..DOES>-word translated into a call of the C function for the DOES>-part, with the address of the C variable produced by the CREATE as top-of-stack parameter.

3.10 Variable Stack Effects

Until now we have had no need to implement a stack in memory or update a stack pointer. All stack items reside in C variables and the translator keeps track of the stacks.

Unfortunately, there are definitions that are not as well-behaved as assumed above. They have control flow joins with unequal stack depths, resulting in a variable stack effect for the whole word. These definitions are rare, but many applications contain one or two of them. If the application requires the translation of all definitions, the translator must handle variable stack effects, too.

While some patterns of variable stack effects can be handled automatically without introducing memory stacks and stack pointers, this is not possible in general. Therefore, we finally have to introduce run-time stacks. Before an unbalanced control structure or a definition with a variable stack effect, all stack items in variables are stored on the stack, inside they are accessed on the stack and the stack pointer is updated, and afterwards the needed stack items are loaded into variables again.

A definition with variable stack effect poisons all its direct and indirect callers. Therefore, the translator should at least recognize common cases (e.g., `?DUP IF...THEN`, where the combined stack effect is fixed) and translate them without introducing-memory stacks.

3.11 Recursive Definitions, `EXECUTE` etc.

Computing the stack effect of a recursive word is a little different from other words: With the normal method we would need the stack effect of the word for computing it. However, if we assume that the recursive word has a fixed stack effect, the stack effect is the same as the stack effect of the non-recursing path(s) through the word. Using this stack effect, the translator can check the validity of the assumption in another pass through the definition. If the definition turns out to have a variable stack effect, it can be translated like any other such definition.

`EXECUTE` and deferred words pose a similar, but harder problem: While all words executed by a specific `EXECUTE` usually have the same stack effect, the translator does not know that stack effect. One solution for this problem is to treat `EXECUTE` and deferred words like words with variable stack effect, the other solution is to use annotations provided by the programmer to specify a stack effect.

Execution tokens are represented by C function pointers, and `EXECUTE` just performs a call to the pointed-to function. This representation of execution tokens implies that the translator has to create a C function for each `CREATE`d word, variable, or constant that is ticked.

3.12 Exception words

`THROW` and `CATCH` can be implemented using `longjmp()` and `setjmp()` without too many problems.

3.13 Cross-compilation problems

One of the more remarkable features of Forth is the removal of the strict division between compile-time and run-time. While typical Forth-to-C translators will have similar restrictions in this respect as some Forth cross-compilers, with some effort these restrictions can be circumvented: Modern operating systems offer dynamic linking of object code. A Forth system based on a Forth-to-C translator could produce C code, and at the end of the definition, invoke the C compiler, and dynamically link the resulting object module. This approach would require an unhealthy amount of patience from the users, but otherwise it would be a full Forth system.

Many implementors of Forth-to-C translators will not want to go to such lengths and will implement the translator in the context of an existing Forth system. In such a setting, compilation and execution can be mixed during the translation stage (i.e., while running on the normal Forth system), while only execution is possible during the execution of the translated and compiled program.

In such cross-translation systems, there's also the problem that addresses cannot be compiled as literals or stored into variables and data structures at compile time, because the addresses are different at run-time. One solution to this problem is to require using typed words (`ALITERAL`, `A!`, `A,` etc.) for these operations, and using the type information for the relocation. A more convenient, but less portable (i.e., OS-dependent) solution is to load the whole image of the translating Forth system to the same address as during the translation. In such a system, variables etc. are not represented as C variables, they behave just like literals.

3.14 Example

A Forth definition of `max ( n1 n2 -- n )` (Fig. 1) is translated into C (Fig. 4), then `gcc-2.6.3 -O3 -fomit-frame-pointer` compiles the C code into Intel assembly (Fig. 5). Note that three of the six instructions here are due to calling overhead and can be eliminated with inlining.

```
Cell max(Cell x2, Cell x1)
{
  Cell x3, x4, x5, x6, x7, x8, x9;

  /* stack now: ... x1 x2 */
  { /* 2dup */
    Cell n1=x1;
    Cell n2=x2;
    x3=n1;
    x4=n2;
    x5=n1;
    x6=n2;
  }
  /* stack now: ... x3 x4 x5 x6 */
  { /* < */
     Cell n1=x3;
     Cell n2=x4;
     Cell n;
     n=FLAG(n1<n2); /* #define FLAG - */
     x7=n;
  }
  /* stack now: ... x3 x4 x7 */
  if (x7==0) goto label1; /* if */
  /* stack now: ... x3 x4 */
  { /* swap */
    Cell n1=x3;
    Cell n2=x4;
    x8=n2;
    x9=n1;
  }
  /* stack now: ... x8 x9 */
  { /* drop */
    Cell n=x9;
  }
  /* stack now: ... x8 */
  goto label2;
label1:
  /* stack now: ... x3 x4 */
  { /* drop */
    Cell n=x4;
  }
  /* stack now: ... x3 */
  x8=x3; /* reconciliation before THEN */
  /* stack now: ... x8 */
label2:
  /* stack now: ... x8 */
  return x8;
}
```

Fig. 4. `max` translated to C

```
  movl 4(%esp),%edx
  movl 8(%esp),%eax
  cmpl %edx,%eax
  jge L5
  movl %edx,%eax
L5:
  ret
```

Fig. 5. `max` in Intel assembly

4 Implementation

The second author has implemented a proof-of-concept Forth-to-C translator in about one man-month, with no prior knowledge of Forth. The translator is based on the Gforth system. The source can be found at http://www.complang.tuwien.ac.at/forth/forth2c.tar.gz (if you only have ftp: ftp.complang.tuwien.ac.at/pub/forth/forth2c.tar.gz).

Basically, the translator hooks itself into some words central to compilation: `COMPILE,`, `LITERAL`, the basic control structure words, `:`, `;`, `CREATE` and friends. It also hooks into words that compile in-line data, like `S"`.

A program is translated by loading it into Gforth. If the translator is turned on, Gforth will not only compile the program into threaded code, but, as a side effect, it will also produce a file containing the C code. In this way, the whole power of Gforth can be used during the translation process.[5]

Since the Forth compiler makes only one pass through a definition, everything must be done in that pass. There is just one problem: The translator must make a pass through the whole definition to compute the stack effect, and it needs to know the stack effect to create the C function header, so it can only start outputting C code at the end of the definition. The solution is to generate the text of the body of the function into a buffer in memory, and write it to the output file at the end of the definition.

The translator must remember the stack effect of a definition somewhere. The best place would be in the header of the definition. Unfortunately, this would require some surgery in the internals of Gforth[6]. Therefore, the translator keeps these informations in a separate wordlist under the same name as the definition. This means, of course, that every

[5] Well, at least in theory; in practice, the translator has a few restrictions described in the file BUGS.F2C.

[6] In particular, in parts that were very hard to understand at the time of the implementation; fortunately, this has improved in the meantime.

definition name must be used only once. But then this is also enforced by the mapping from Forth names to C names (which does not take wordlist membership into account and C has a flat name space).

In contrast to the suggestion in Section 3.3, our translator reuses stack item variables. A variable represents a stack item with a certain offset from the stack bottom during the whole function. E.g., the stack item on the top-of-stack upon function entry is called `p0`, the item below it `p1` etc.; the stack item above `p0` is called `x0`, the next one `x1` etc. The advantage of this scheme is simplicity: the translator need not keep track of an unlimited number of names, a few counters are sufficient, and most of these counters are needed anyway, to keep track of the stack depth.

This scheme also makes it trivial to reconcile the stack state at control flow joins: The respective stack items are already in the same variables, so there is nothing that needs reconciling. As a result, the control flow stack contains relatively simple items: In addition to the normal control flow information, it stores the label number and the stack depths at the point where the control flow item was generated.[7] Our translator stores this extended control flow information on a separate, user-defined stack (Gforth stores its information on the data stack).

While it would have been nice to generate the translation of primitives automatically from Gforth's primitive specifications, as suggested in Section 3.3, the translator does not employ this approach, but uses hand-written primitives translation.

The most important limitations of this prototype translator are that it cannot translate words defined with `DOES>`, initialized created words, `EXECUTE`, many recursive definitions, definitions with variable stack effects or definitions that use the floating-point stack or locals. Some of these restrictions would be easy to remove, some of them would take more effort.

If you wonder how the translator handles the `max` example above, the generated C code looks somewhat different, mainly due to the reuse of variables, but it is compiled to the same assembly code (see Fig. 5).

5 Empirical Results

Due to the restrictions of our translator we could not use it on realistically sized benchmarks, only on small ones. On the other hand, this restriction to small benchmarks means that it is easier to compare with other Forth systems.

The benchmarks we used were the ubiquitous Sieve (counting the primes < 16384 a thousand times); bubble-sorting (6000 integers) and matrix multiplication (200×200 matrices) come from the Stanford integer benchmarks (originally in Pascal, but available in C[8]) and have been translated into Forth by Martin Fraeman and included in the TILE Forth package. These three benchmarks share one disadvantage: They have an unusually low amount of calls (in typical Forth code, every third or fourth executed word is a call or return). To benchmark calling performance, we computed the 34^{th} fibonacci number using a recursive algorithm with exponential run-time complexity.

Most measurements were made under Linux; bigForth and iForth were benchmarked under DOS/GO32; Win32Forth, NT Forth and its NCC under Windows NT. The Windows NT systems were run on one machine, iForth on a different machine, and the rest on another machine; all three machines had a 486DX2/66 with 256K secondary cache and are similar in performance. We used the median of three measurements of the user time (the system time was negligible anyway). Of course, we measured the output of our prototype translator (compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`) and the output of the Forth-to-C translator included in the Timbre V.4 distribution (compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`). To illustrate our point that there is no such thing as efficient code per se, we compiled the output of our translator also with GCC without optimizations. And to see how well our translator does compared to a human, we also compare with hand-coded C programs (the original C versions in case of the Stanford benchmarks). Traditional Forth native code generation techniques are represented by bigForth 386 (v1.20β) [Pay91], by iForth 1.06 and by LMI's NT Forth NCC. Finally, we measured some interpretive systems: Gforth (indirect threaded code, compiled with GCC 2.6.3, default flags and `-DFORCE\_REG`), Win32Forth 1.2093, NT Forth (beta, May 1994), PFE-0.9.11 (compiled with GCC 2.6.3 with the default configuration), and ThisForth Beta (compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`). We use Gforth (a fast system based on an engine written in GNU C [Ert93]) as a reference point in the following discussion.

Figure 6 shows the time that the output of our translator needs for the benchmarks, and the time

[7] Our translator is somewhat buggy in this area, it does not handle `THEN` correctly.

[8] The C version and Martin Fraeman's original translations to Forth can be found at ftp.complang.tuwien.ac.at/pub/forth/stanford-benchmarks.tar.gz

relative time	f2c opt.	Timbre	f2c no opt.	hand-coded C	big-Forth	iForth	NT F. NCC	Gforth	Win32-Forth	NT Forth	PFE	This-Forth	abs. time f2c opt.
sieve	1.00		7.03	0.86	1.87	2.14	1.27	6.15	7.99	6.56	10.26	18.32	5.19s
bubble	1.00		8.28	0.87	2.34	2.91	7.12	7.43	9.69	10.41	12.37		4.79s
matmul	1.00		9.35	1.10	3.02	2.35	4.14	7.04	9.87	9.09	15.80		4.02s
fib	1.00	3.14	4.92	1.00	1.37	1.32	1.47	4.61	6.62	5.81	8.40	12.99	7.96s

Fig. 6. Time needed by various systems for several benchmarks, relative to the output of our prototype Forth-to-C translator (f2c opt.); lower means faster.

other systems take relative to this baseline (or, in other words, the speedup factor that our translator (and GCC) achieves over the other systems. Empty entries indicate that we did not succeed in running the benchmark on the system. The combination of our translator and an optimizing GCC is 4.5–7.5 times faster than Gforth. We achieved similar results on a DecStation 5000/150 (50/100MHz R4000), where the first three benchmarks were 6–8 times faster than Gforth. As expected, the speedup for the Fibonacci benchmark is not as large as for the other benchmarks, but, contrary to a myth popular in the Forth community, calls in C are not slow: The Fibonacci benchmark translated to C is almost five times faster than in Gforth, and it even beats Forth native code compilers.

The result of Timbres Forth-to-C translator is slow, as expected (since Timbre does not have `DO..LOOP` and friends, we could only measure fib, but we think this result is representative). Combining our translator with a non-optimizing GCC results in code that is even slower than the interpretive Gforth system, confirming the points we make about efficient code in Section 3.1. Hand-coded C is between 14% faster and 10% slower than the output of the Forth translator. We were a little surprised by the matrix multiplication result, where the code translated to Forth and back was faster than the original. Closer inspection showed that, in translating from C to Forth, an optimization had been performed, which the C compiler did not perform (it is an interprocedural optimization that requires interprocedural alias analysis) and which reduced the amount of memory accesses; this is probably responsible for the speedup, maybe combined with vagaries such as instruction cache alignment.

BigForth and iForth achieve a speedup of about 3 over Gforth, but there is still a lot of room for improvement (at least a factor of 1.3–3). The results of NT Forth NCC are a bit worse on average and have a big variance (a speedup of 1–5 over Gforth, 1.2–7.2 times slower than f2c opt.). These results show that researching better native code generation techniques is not just a waste of time and that there is still a lot to be gained in this area. These results also show that the following statement has not become outdated yet: "The resulting machine code is a factor of two or three slower than the equivalent code written in machine language, due mainly to the use of the stack rather than registers." [Ros86]

Win32Forth is 6.5–10 times slower than the translated and optimized code, NT Forth 5.5–10.5 times, PFE 8–16 times, and ThisForth 13–18 times (on the benchmarks that worked). The slowdown of PFE and ThisForth with respect to Gforth can be explained with the self-imposed restriction to standard C (although the measured configuration of PFE uses a GNU C extension: global register variables), which makes efficient threading impossible. Moreover, current C compilers have a hard time optimizing other aspects of the ThisForth source. The performance of Win32Forth and NT Forth relative to Gforth may surprise you (at least it surprised us); after all, these systems are written in assembly language and therefore enjoy some benefits—in particular, they can make better use of the registers. One important reason for the disappointing performance of these systems is probably that they are not written optimally for the 486 (e.g., they use `lods`).[9] In addition, Win32Forth uses a comfortable, but costly method for relocating the Forth image: it computes the actual addresses at run time, resulting in two address computations per NEXT.

	interp. size	.o size	size ratio	compile time	source lines	C lines
sieve	418	272	1.54	1.10s	25	482
bubble	1020	748	1.36	1.60s	72	1100
matmul	784	412	1.90	1.40s	55	793
fib	140	140	1.00	0.90s	10	169

Fig. 7. Code size and compile time

We not only measured run-time, but also code size and compile time (see Fig. 7). For threaded code (interp. size), we measured the space alloted in the Gforth system during compilation of the pro-

[9] For comparison: A system fragment written in assembly language and hand-tuned for the 486 is about 20% faster on the Sieve than the measured configuration of Gforth [Beu95].

gram and subtracted the alloted data; i.e., the interpreted code size includes the space needed for headers. Gforth uses one cell (32 bits in the measured system) per compiled word, two cells for the code field and padding in the header such that the body is maximally (i.e., 8-byte-) aligned. For the size of the machine code produced by the translator/compiler combination (.o size), we used the sum of the text and data sizes produced by the Unix `size` command, as applied to the object (`.o`) file. This does not include the size of the symbol table information included in the object file (which is easy to strip away after linking). The data size in the object file does not include the alloted space, as that is allocated later at run-time.

The code size measurements dispell another popular myth, that of the inherent size advantage of stack architecture code and of the bloat produced by optimizing C compilers. While a comparison of a header-stripping 16-bit Forth with a RISC (≈50% bigger code than CISCs) would give a somewhat different result, the observed size differences of more than an order of magnitude need a different explanation: differences in the functionality of the software and different software engineering practices come to mind.

For the compile time measurements, Fig. 7 only displays the user time needed by GCC to compile and link the program. The system time was constant at 0.6s. The compilation to Gforth's interpreted code needed a negligible amount of time; the translation to C also vanished in the measurement noise, although it was not written for speed and although the present implementation should be much slower than normal Forth compilation. The compile time data indicate that, after a startup time of about 1.4s (user+system), GCC compiles about 90 lines of Forth code (1500 lines of translator output) per second. Interestingly, less than one byte of machine code is generated per line of C code.

6 Conclusion

The secret of efficient C code for current optimizing C compilers is to do as much as possible in local variables. This paper presents a method for translating Forth into such C code. A significant subset of Forth can even be translated into C code where all stack items are kept in local variables, and the stack pointer and all operations on it are unnecessary. This method is so simple that someone new to Forth could implement it in a person-month. This translation method combined with an optimizing C compiler achieves impressive speedups over interpretive Forth implementations (4.5–18), over a different Forth-to-C translator (3.1), and even over commercial native code compilers (1.2–7.2).

Acknowledgements

Martin Fraeman provided the C versions of the Stanford benchmarks. Kenneth O'Heskin kindly produced the benchmark results for Win32Forth, NT Forth, and NT Forth NCC. Marcel Hendrix provided the iForth results. The comments of Manfred Brockhaus helped improve the paper.

References

[Alm86] Thomas Almy. Compiling Forth for performance. *Journal of Forth Application and Research*, 4(3):379–388, 1986.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990.

[Beu95] Bernd Beuster. Usenet posting in de.comp.lang.forth, May 1995.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[Ert92] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, pages 73–78, Southampton, England, 1992. MicroProcessor Engineering.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Lázně (Marienbad), 1993.

[Ert94] M. Anton Ertl. Automatic scoping of local variables. In *EuroForth '94 Conference Proceedings*, pages 31–37, Winchester, UK, 1994.

[FH91] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[Pay91] Bernd Paysan. Ein optimierender Forth-Compiler. *Vierte Dimension*, 7(3):22–25, September 1991.

[Ros86] Anthony Rose. Design of a fast 68000-based subroutine-threaded Forth with inline code & an optimizer. *Journal of Forth Application and Research*, 4(2):285–288, 1986. 1986 Rochester Forth Conference.

[Smi92] Norman Smith. *Write Your Own Programming Language Using C++*. Wordware Publishing, 1992. ISBN 1-55622-264-5.